

# Domain Views for Constraint Programming

P. Van Hentenryck<sup>1</sup> and L. Michel<sup>2</sup>

<sup>1</sup> NICTA, Australia, RI 02912

<sup>2</sup> University of Connecticut, Storrs, CT 06269-2155

**Abstract.** Views are a standard abstraction in constraint programming: They make it possible to implement a single version of each constraint, while avoiding to create new variables and constraints that would slow down propagation. Traditional constraint-programming systems provide the concept of *variable views* which implement a view of the type  $y = f(x)$  by delegating all (domain and constraint) operations on variable  $y$  to variable  $x$ . This paper proposes the alternative concept of *domain views* which only delegate domain operations. Domain views preserve the benefits of variable views but simplify the implementation of value-based propagation. Domain views also support non-injective views compositionally, expanding the scope of views significantly. Experimental results demonstrate the practical benefits of domain views.

## 1 Introduction

Constraint programming systems provide rich libraries of constraints, each of which models some specific structure useful across a wide range of applications. These constraints are important both from a modeling standpoint, as they make it possible to state problems at a high level of abstraction, and from an efficiency standpoint, as they allow dedicated algorithms to exploit the specific structure. The global constraint catalog [2] in fact lists about 354 global constraints at the time of writing. In addition, each of these constraints potentially come in many different forms as they can be applied, not only on variables, but also on expressions involving variables.

This large number of variants presents a challenge for system developers who must produce, validate, optimize, and maintain each version of each constraint. To avoid the proliferation of such variants, system developers often prefer to design a unique variant over variables and introduce new variables and constraints to model the more complex cases. For instance, a constraint

$$alldifferent(x_1 + 1, \dots, x_n + n)$$

can be modeled by a system of constraints

$$\{alldifferent(y_1, \dots, y_n), y_1 = x_1 + 1, \dots, y_n = x_n + n\}$$

where the  $y_i$ 's are new variables. This approach keeps the system core small but introduces an overhead in time and space. Indeed, the new constraints must

be propagated through the constraint engine and the system must maintain additional domains and constraints, increasing the cost of propagation and the space requirements.

Over the years, system designers have sought ways to mitigate this difficulty and proposed several solutions of varying complexity. Prolog-style languages offered indexicals [5,3] while C++ libraries like Ilog Solver [6] introduces the concept of variable views. For an injective function  $f$  and a variable (or a view)  $x$ , a variable view  $y$  enforces the equivalent of the constraint  $y = f(x)$  but it does not introduce a new variable and a new constraint: Instead, it delegates all domain and constraint operations (the ability to wake constraints) on  $y$  to  $x$ , sometimes after applying  $f^{-1}$ . These variable views remove the time and space overhead mentioned above and keep the solver kernel small, thus giving us a valuable abstraction for constraint programming. Recently, [8,9] demonstrated how variable views can be implemented in terms of C++ templates, providing further improvement in speed and memory usage. The idea is to use parametric polymorphism to allow for code reuse and compile-time optimizations based on code expansion and inlining. [9] demonstrates that variable views provide significant software engineering benefits as well as great computational improvements over the basic approach using new variables and constraints.

This paper aims at expanding the scope of constraint-programming views with an extremely simple abstraction: The concept of *domain views* which only delegate domain operations. Domain views preserve the benefits of variable views but simplify the implementation of value-based propagation, i.e., the propagation of events of the form  $\langle c, x, v \rangle$ , meaning that constraint  $c$  must be propagated because variable  $x$  has lost value  $v$  (e.g., [10,4]). The key benefit of domain views is to support *non-injective views* elegantly and compositionally. Domain views can also be implemented using parametric polymorphism and hence are fully compatible with the compilation techniques in [9].

The rest of the paper is organized as follows. Sections 2, 3, and 4 present the preliminaries on constraint programming and on views. Section presents the implementation of variable views. Section 5 introduces the concept of domain views. Section 6 demonstrates how to generalize domain views to the case where the function  $f$  is not injective. Section 7 briefly discusses how to exploit monotonicity and anti-monotonicity. Section 8 presents experimental results. Section 9 discusses related work on advisors [7] and Section 10 concludes the paper.

## 2 Preliminaries

A constraint-programming system is organized around a queue of events  $\mathcal{Q}$  and its main component is an engine propagating constraints in the queue, i.e.,

---

```

1 while  $\neg \text{empty}(\mathcal{Q})$  do
2    $\text{propagate}(\text{pop}(\mathcal{Q}))$ ;

```

---

For simplicity, we only consider two types of events:  $\langle c, x \rangle$  and  $\langle c, x, v \rangle$ . An event  $\langle c, x \rangle$  means that constraint  $c$  must be propagated because the domain of variable

---

```

1 interface Variable
2   bool member( $\mathcal{V}$  v);
3   bool remove( $\mathcal{V}$  v);
4   void watch( $\mathcal{C}$  c);
5   void watchValue( $\mathcal{C}$  c);
6   void wake;
7   void wakeValue( $\mathcal{V}$  v);

```

---

**Fig. 1.** The Variable Interface.

$x$  has been shrunk. An event  $\langle c, x, v \rangle$  means that constraint  $c$  must be propagated because the value  $v$  has been removed from the domain of variable  $x$ . Events of the form  $\langle c, x \rangle$  are sometimes called *variable-based propagation*, while those of the form  $\langle c, x, v \rangle$  are sometimes called *value-based propagation*. Note that some systems also implement what is called *constraint-based propagation*, where the event simply consists of constraint to propagate without additional information. We do not discuss constraint-based propagation here since it is easier to handle.

The propagation of a constraint may change the domains of some variables and thus introduce new events in the queue. As a result, a variable  $x$  not only maintains its domain  $D(x)$  but also keeps track of the constraints it appears in so that the proper events can be inserted in the queue. As a result, a variable  $x$  is best viewed as a triple  $\langle D, SC, SC_v \rangle$ , where  $D$  is the domain of the variable,  $SC$  is the set of constraints involving  $x$  that use variable-based propagation, and  $SC_v$  is the set of constraints involving  $x$  that use value-based propagation. If  $x$  is a variable, we use  $D(x)$ ,  $SC(x)$  and  $SC_v(x)$  to denote these three components.

For simplicity, a variable in this paper implements the interface depicted in Figure 1, where  $\mathcal{V}$  denotes the set of values considered (e.g., integers or reals) and  $\mathcal{C}$  the set of constraints. For a variable  $x$ , method **member**( $v$ ) tests  $v \in D(x)$ , method **remove**( $v$ ) implements  $D(x) := D(x) \setminus \{v\}$  and returns true if the resulting domain is not empty, method **watch**( $c$ ) registers constraint  $c$  for variable-propagation, and method **watchValue**( $c$ ) registers constraint  $c$  for value-propagation. The **wake** methods are used for creating new events in the queue. Method **wake** must implement  $\mathcal{Q} := \mathcal{Q} \cup \{\langle c, x \rangle \mid c \in SC(x)\}$  while method **wakeValue**( $v$ ) must implement  $\mathcal{Q} := \mathcal{Q} \cup \{\langle c, x, v \rangle \mid c \in SC_v(x)\}$ . With our conventions, a variable can be implemented as depicted in Figure 2.

### 3 Views

The purpose of this paper is to define and implement abstractions for constraints of the form  $y = \psi(x)$ . In a first step, the paper focuses on injective views, i.e., views in which function  $\psi$  is injective, which is the functionality provided by many constraint-programming solvers.

**Definition 1 (Injective Function)** *A function  $\psi : D \rightarrow \mathcal{V}$  is injective if*

$$\forall v, v' \in D : \psi(v) = \psi(v') \Rightarrow v = v'.$$

---

```

1 implementation DomainVariable
2   {V} D;
3   {C} SC;
4   {C} SCv;
5
6 DomainVariable({V} Do) { D := Do; SC := ∅; SCv := ∅; }
7 bool member(V v) { return v ∈ D; }
8 bool remove(V v) {
9   if v ∈ D
10    D := D \ {v};
11    wake();
12    wakeValue(v);
13 }
14 void watch(C c) { SC := SC ∪ {c}; }
15 void watchValue(C c) { SCv := SCv ∪ {c}; }
16 void wake() { Q := Q ∪ {⟨c, this⟩ | c ∈ SC}; }
17 void wakeValue(V v) { Q := Q ∪ {⟨c, this, v⟩ | c ∈ SCv}; }

```

---

**Fig. 2.** The Implementation of a Domain Variable

The inverse  $\psi^{-1} : \mathcal{V} \rightarrow D_{\perp}$  of injective function  $\psi$  is defined as

$$\psi^{-1}(w) = \begin{cases} v & \text{if } v \in D \wedge \psi(v) = w \\ \perp & \text{otherwise} \end{cases}$$

where  $D_{\perp} = D \cup \{\perp\}$ .

Note that the definition of  $\psi^{-1}$  is a specification: An actual implementation uses a dedicated implementation of  $\psi^{-1}$  as the following two examples illustrate.

**Example 1 (Shift View)** Consider the view  $y = x + c$  where  $c$  is an integer and  $x$  and  $y$  are integer variables. Function  $\psi : \mathbb{Z} \rightarrow \mathbb{Z}$  can be specified (using lambda calculus notation [1]) as  $\lambda k.k + c$ . Its inverse  $\psi^{-1} : \mathbb{Z} \rightarrow \mathbb{Z}$  is defined as  $\lambda k.k - c$ .

**Example 2 (Affine View)** Consider the view  $y = ax + b$  where  $a, b \in \mathbb{Z}$  and  $x, y$  are integer variables.  $\psi : \mathbb{Z} \rightarrow \mathbb{Z}$  is  $\lambda k.ak + b$ . Its inverse  $\psi^{-1} : \mathbb{Z} \rightarrow \mathbb{Z}$  is

$$\psi^{-1} = \begin{cases} \lambda k.(k - b)/a & \text{if } (k - b) \bmod a = 0 \\ \lambda k.\perp & \text{otherwise.} \end{cases}$$

Views must be compositional and make it possible to state a view over a view.

## 4 Variable Views

The fundamental idea of variable views, implemented in many systems, is to delegate all domain and constraint operations of variable  $y$  to variable  $x$ . A variable view thus implements an *adapter pattern* that stores neither domain

---

```

1 implementation DomainVariable
2   { $\mathcal{V}$ }  $D$ ;
3   { $\langle \mathcal{C}, \mathcal{X} \rangle$ }  $SC$ ;
4   { $\langle \mathcal{C}, \mathcal{X}, \mathcal{F} \rangle$ }  $SC_v$ ;
5
6 DomainVariable({ $\mathcal{V}$ }  $D_o$ ) {  $D := D_o$ ;  $SC := \emptyset$ ;  $SC_v := \emptyset$ ; }
7 bool member( $\mathcal{V}$   $v$ ) { return  $v \in D$ ; }
8 bool remove( $\mathcal{V}$   $v$ ) {
9   if  $v \in D$ 
10      $D := D \setminus \{v\}$ ;
11     wake();
12     wakeValue( $v$ );
13 }
14 void watch( $\mathcal{C}$   $c$ ,  $\mathcal{X}$   $y$ ) {  $SC := SC \cup \{\langle c, y \rangle\}$ ; }
15 void watchValue( $\mathcal{C}$   $c$ ,  $\mathcal{X}$   $y$ ,  $\mathcal{F}$   $\psi$ ) {  $SC_v := SC_v \cup \{\langle c, y, \psi \rangle\}$ ; }
16 void watch( $\mathcal{C}$   $c$ ) { watch( $c$ , this); }
17 void watchValue( $\mathcal{C}$   $c$ ) { watch( $c$ , this,  $\lambda k.k$ ); }
18
19 void wake() {  $\mathcal{Q} := \mathcal{Q} \cup \{\langle c, x \rangle \mid \langle c, x \rangle \in SC\}$ ; }
20 void wakeValue( $\mathcal{V}$   $v$ ) {  $\mathcal{Q} := \mathcal{Q} \cup \{\langle c, x, \psi(v) \rangle \mid \langle c, x, \psi \rangle \in SC_v\}$ ; }

```

---

**Fig. 3.** The Domain Variable for Variable Views.

nor sets of constraints. The variable view simply stores a reference to variable  $x$  and delegates all domain and constraint operations to  $x$ , possibly after applying function  $\psi$  or  $\psi^{-1}$  on the arguments. Informally speaking, the membership test  $w \in D(y)$  becomes  $\psi^{-1}(w) \in D(x)$ , the removal operation proceeds similarly and variable  $x$  also watches all the constraints of  $y$ .

The only difficulty in variable views comes from the fact that variable  $x$  now needs to watch constraints on both  $x$  and  $y$ . For variable-based propagation, it is necessary to remember which variable is being watched for each constraint and the set  $SC$  now consists of pairs  $\langle c, z \rangle$  where  $c$  is a constraint and  $z$  is a variable. For value-based propagation, it is necessary to store the function  $\psi$  since it must be applied when method **wakeValue** is applied. Hence the set  $SC_v$  now contains triples of the form  $\langle c, z, \psi \rangle$ . These generalizations are necessary, since when a value  $v$  is removed from the domain of  $x$ , the value-based events for variable  $y$  must be of the form  $\langle c, y, \psi(v) \rangle$ .

The implementation of variables to support variable views is shown in Figure 3 where  $\mathcal{X}$  denotes the set of variables/views and  $\mathcal{F}$  the set of first-order functions. Observe the types of  $SC$  and  $SC_v$  in lines 3–4, the new methods in lines 14–15 allow to watch a constraint  $c$  for a view  $y$ , the matching redefinition of the **watch** methods, and the **wake** methods that store additional information in the queue by applying the stored function  $\psi$  on value  $v$  (line 20).

Figure 4 depicts a template for variable views in terms of an injective function  $\psi$ . A shift view specialization is shown in Figure 5. Observe that variable views do not store a domain nor constraint sets. Methods **member** and **remove** apply  $\psi^{-1}$  as mentioned earlier with only the addition of a test for the  $\perp$  case. Methods **watch**

---

```

1 implementation VariableView< $\psi$ >
2    $\mathcal{X}$  x;
3 VariableView( $\mathcal{X}$   $\underline{x}$ ) {  $x := \underline{x}$ ; }
4 bool member( $\mathcal{V}$   $v$ ) {
5   if  $\psi^{-1}(v) \neq \perp$  return  $x.\text{member}(\psi^{-1}(v))$ ; else return false;
6 }
7 bool remove( $\mathbb{Z}$   $v$ ) {
8   if  $\psi^{-1}(v) \neq \perp$  return  $x.\text{remove}(\psi^{-1}(v))$ ; else return true;
9 }
10 void watch( $\mathcal{C}$   $c, \mathcal{X}$   $y$ ) {  $x.\text{watch}(c, y)$ ; }
11 void watchValue( $\mathcal{C}$   $c, \mathcal{X}$   $y, \mathcal{F}$   $\phi$ ) {  $x.\text{watchValue}(c, y, \phi \circ \psi)$ ; }
12 void watch( $\mathcal{C}$   $c$ ) {  $x.\text{watch}(c, \text{this})$ ; }
13 void watchValue( $\mathcal{C}$   $c$ ) {  $x.\text{watchValue}(c, \text{this}, \psi)$ ; }

```

---

**Fig. 4.** The Template for Variable Views.

---

```

1 implementation VariableShiftView
2    $\mathcal{X}$  x;
3    $\mathbb{Z}$  c;
4 VariableShiftView( $\mathcal{X}$   $\underline{x}, \mathbb{Z}$   $\underline{c}$ ) {  $x := \underline{x}; c := \underline{c}$ ; }
5 bool member( $\mathbb{Z}$   $v$ ) { return  $x.\text{member}(v - c)$ ; }
6 bool remove( $\mathbb{Z}$   $v$ ) { return  $x.\text{remove}(v - c)$ ; }
7 void watch( $\mathcal{C}$   $c, \mathcal{X}$   $y$ ) {  $x.\text{watch}(c, y)$ ; }
8 void watchValue( $\mathcal{C}$   $c, \mathcal{X}$   $y, \mathbb{Z} \rightarrow \mathbb{Z}$   $\phi$ ) {  $x.\text{watchValue}(c, y, \phi \circ (\lambda k.k + c))$ ; }
9 void watch( $\mathcal{C}$   $c$ ) {  $x.\text{watch}(c, \text{this})$ ; }
10 void watchValue( $\mathcal{C}$   $c$ ) {  $x.\text{watchValue}(c, \text{this}, \lambda k.k + c)$ ; }

```

---

**Fig. 5.** A Variable View for Shift Views.

and **watchValue** (lines 10–11) state a view on the view itself. In particular, line 11 illustrates the need for function composition in the case of value propagation.

The instantiation for shift views in Figure 5 highlights some interesting points. First, there is no need for a  $\perp$  test, since the inverse of  $\psi$  is always in the domain of  $\psi$ . Second, value-based propagation requires the use of first-order functions (see lines 9 and 12) or objects implementing the same functionalities. In contrast, methods **member** and **remove** “inline” function  $\phi^{-1}$  in the code, which is never stored or passed as a parameter.

*Optimization* Variable views now stores tuples  $\langle c, z, \psi \rangle$  for value-based propagation. Observe however that  $z$  is an object so that it is possible to use it to compute function  $\psi$ . This only requires the view to provide a method **map** that maps the value  $v$  through  $\psi$ . Lines 15 and 20 in Figure 3 become

---

```

1 void watchValue( $\mathcal{C}$   $c, \mathcal{X}$   $y$ ) {  $SC_v := SC_v \cup \{\langle c, y \rangle\}$ ; }
2 void wakeValue( $\mathcal{V}$   $v$ ) {  $\mathcal{Q} := \mathcal{Q} \cup \{\langle c, x, x.\text{map}(v) \rangle \mid \langle c, x \rangle \in SC_v\}$ ; }

```

---

The **map** method on standard variables is defined as

---

```

1  $\mathcal{V}$  map( $\mathcal{V}$   $v$ ) { return  $v$ ; }

```

---

and its definition on views (defined over variable  $x$  with injective function  $\psi$ ) is

---

```

1 implementation DomainVariable
2   {V} D;
3   {C} SC;
4   {C} SCv;
5   {X} Views;
6
7 DomainVariable({V} Do) { D := Do; SC := ∅; SCv := ∅; Views := ∅; }
8 void addView(X x) { Views := Views ∪ {x}; }
9 bool member(V v) { return v ∈ D; }
10 bool remove(V v) {
11   if v ∈ D
12     D := D \ {v};
13     wake();
14     wakeValue(v);
15     for all y ∈ Views
16       y.wake();
17       y.wakeValue(v);
18 }
19 void watch(C c) { SC := SC ∪ {c}; }
20 void watchValue(C c) { SCv := SCv ∪ {c}; }
21 void wake() { Q := Q ∪ {⟨c, this⟩ | c ∈ SC}; }
22 void wakeValue(V v) { Q := Q ∪ {⟨c, this, v⟩ | c ∈ SCv}; }

```

---

**Fig. 6.** The Domain Variable for Domain Views.

---

```

1 V map(V v) { return ψ(x.map(v)); }

```

---

Observe the recursive call, since views can be posted on views. This optimization clutters a bit the API of variables and views but only minimally.

Variable views are an important concept in constraint programming for injective functions. For constraint-based and variable-based propagation, the implementation is simple and efficient, although it requires to upgrade slightly the data structure to watch constraints. For value-based propagation, the implementation is a bit more cumbersome. It requires a generalization of the constraint queue and the addition of a **map** method on variables and views to avoid manipulating first-order functions. Domain views provide an extremely simple alternative, which also has the benefits of supporting non-injective functions elegantly.

## 5 Domain Views

The key idea behind domain views is to delegate only domain operations from variable  $y$  to variable  $x$ : The view for  $y$  maintains its own constraints to watch. This removes the need to manipulate first-order functions. To implement domain views, traditional variables (and views) must store which variables are viewing them. When their domains change, they must notify their views.

Figure 6 depicts the revised implementation of domain variables to support domain views. The variable now keeps its views (line 5) and provides a method

---

```

1 implementation DomainView< $\psi$ >
2    $\mathcal{X}$   $x$ ;
3    $\{C\}$   $SC$ ;
4    $\{C\}$   $SC_v$ ;
5    $\{X\}$   $Views$ ;
6 DomainView( $\mathcal{X}$   $x$ ) {  $SC := \emptyset$ ;  $SC_v := \emptyset$ ;  $Views := \emptyset$ ; }
7 void addView( $\mathcal{X}$   $x$ ) {  $Views := Views \cup \{x\}$ ; }
8 bool member( $\mathcal{V}$   $v$ ) {
9   if  $\psi^{-1}(v) \neq \perp$  return  $x.member(\psi^{-1}(v))$ ; else return false;
10 }
11 bool remove( $\mathbb{Z}$   $v$ ) {
12   if  $\psi^{-1}(v) \neq \perp$  return  $x.remove(\psi^{-1}(v))$ ; else return true;
13 }
14 void watch( $C$   $c$ ) {  $SC := SC \cup \{c\}$ ; }
15 void watchValue( $C$   $c$ ) {  $SC_v := SC_v \cup \{c\}$ ; }
16 void wake() {
17    $\mathcal{Q} := \mathcal{Q} \cup \{ \langle c, this \rangle \mid c \in SC \}$ ;
18   forall ( $y \in Views$ )  $y.wake()$ ;
19 }
20 void wakeValue( $\mathcal{V}$   $v$ ) {
21    $\mathcal{Q} := \mathcal{Q} \cup \{ \langle c, this, v \rangle \mid c \in SC_v \}$ ;
22   forall ( $y \in Views$ )  $y.wakeValue(\psi(v))$ ;
23 }

```

---

**Fig. 7.** The Template for Domain Views.

for adding a view (line 8). The only other change is in method **remove** in lines 16–18: The domain variable calls method **wake** and **wakeValue** on its views to inform them of the loss of value  $v$  to let them schedule their own constraints.

Figure 7 shows a template for domain views in terms of an injective function  $\psi$ . A specialization for shift views is shown in Figure 8. Observe first how the domain view maintains its own set of constraints. It delegates its domain operations in methods **member** and **remove** in the same way as variable views, but it does not delegate its **watch** methods, which are similar to those of a traditional domain variable. To implement views on views, the **wake** methods also wake the views (lines 16–20 and 21–25), using the function  $\psi$  to send the appropriate value since  $v$  is the value removed from  $D(x)$ .  $D(x)$  may be explicit (traditional variable) or implicit (views). The shift view in Figure 8 does not manipulate first-order functions and inlines  $\psi^{-1}$  in lines 9 and 12 and  $\psi$  in line 24.

Domain views provide an elegant alternative to variable views. They remove the need to modify the data structure for watching constraint and alleviate the need for the **map** function, while preserving the benefits of variable views and enabling more inlining for value-based propagation. They are based on a simple idea: Only delegating the domain operations. Instead of delegating constraint watching, constraints are watched locally. It is interesting to analyze the memory requirements of both approaches. Variable views need to store variables in their constraint lists, which require space proportional to the length of these lists. In contrast, domain views only require a few pointers for their own lists,



---

```

1 implementation DomainShiftView
2    $\mathcal{X} \ x;$ 
3    $\{C\} \ SC;$ 
4    $\{C\} \ SC_v;$ 
5    $\{\mathcal{X}\} \ Views;$ 
6    $\mathbb{Z} \ c;$ 
7 DomainShiftView( $\mathcal{X} \ x, \mathbb{Z} \ c$ ) {
8    $SC := \emptyset; SC_v := \emptyset; Views := \emptyset; c := c;$ 
9 }
10 bool member( $\mathbb{Z} \ v$ ) { return  $x.member(v-c);$  }
11 bool remove( $\mathbb{Z} \ v$ ) { return  $x.remove(v-c);$  }
12 void watch( $C \ c$ ) {  $SC := SC \cup \{c\};$  }
13 void watchValue( $C \ c$ ) {  $SC_v := SC_v \cup \{c\};$  }
14 void wake() {
15    $\mathcal{Q} := \mathcal{Q} \cup \{ \langle c, this \rangle \mid c \in SC \};$ 
16   for all ( $y \in Views$ )  $y.wake();$ 
17 }
18 void wakeValue( $\mathcal{V} \ v$ ) {
19    $\mathcal{Q} := \mathcal{Q} \cup \{ \langle c, this, v \rangle \mid c \in SC_v \};$ 
20   for all ( $y \in Views$ )  $y.wakeValue(v+c);$ 
21 }

```

---

**Fig. 8.** A Domain View for Shift Views.

the constraints themselves being present in both approaches albeit in different lists. The viewed variables must also maintain the list of its views, which is proportional to the number of views.

## 6 Non-injective Views

We now generalize domain views to non-injective functions.

**Definition 2 (Inverse of a Non-Injective Function)** *The inverse  $\psi^{-1} : \mathcal{V} \rightarrow 2_{\perp}^D$  of non-injective function  $\psi : D \rightarrow \mathcal{V}$  is defined as*

$$\psi^{-1}(w) = \begin{cases} \perp & \text{if } \nexists v \in D : \psi(v) = w \\ \{v \in D \mid \psi(v) = w\} & \text{otherwise.} \end{cases}$$

Figure 9 gives the template for non-injective views. There are only a few modifications compared to the template for injective views. The **member** function must now test membership for a set of values (line 12) and the **remove** function must remove a set of values (line 18). Finally, method **wakeValue(w)** must test membership of  $v = \psi(w)$ , since there may be multiple supports for  $v$  in  $D(x)$ .

The key advantage of domain views is that they own their constraints. In the context of non-injective functions, this is critical since only the view “knows” whether its constraints must be scheduled for propagation.

It is more difficult and less elegant, but not impossible, to generalize variable views to support non-injective functions. Consider what should happen for

---

```

1 implementation NonInjectiveDomainView < $\psi$ >
2    $\mathcal{X}$   $x$ ;
3    $\{C\}$   $SC$ ;
4    $\{C\}$   $SC_v$ ;
5    $\{X\}$   $Views$ ;
6 NonInjectiveDomainView( $\mathcal{X}$   $x$ ) {
7    $SC := \emptyset$ ;  $SC_v := \emptyset$ ;  $Views := \emptyset$ ;
8 }
9 void addView( $\mathcal{X}$   $x$ ) {  $Views := Views \cup \{x\}$ ; }
10 bool member( $\mathcal{V}$   $v$ ) {
11   if  $\psi^{-1}(v) \neq \perp$  return  $\exists w \in \psi^{-1}(v) : x.member(w)$ ;
12   else return false;
13 }
14 bool remove( $\mathcal{V}$   $v$ ) {
15   if  $\psi^{-1}(v) \neq \perp$ 
16     forall( $w \in \psi^{-1}(v)$ ) if  $\neg x.remove(w)$  return false;
17   return true;
18 }
19 void watch( $C$   $c$ ) {  $SC := SC \cup \{c\}$ ; }
20 void watchValue( $C$   $c$ ) {  $SC_v := SC_v \cup \{c\}$ ; }
21 void wake() {
22    $\mathcal{Q} := \mathcal{Q} \cup \{ \langle c, this \rangle \mid c \in SC \}$ ;
23   forall( $y \in Views$ )  $y.wake()$ ;
24 }
25 void wakeValue( $\mathcal{V}$   $w$ ) {
26    $v = \psi(w)$ ;
27   if  $x.member(v)$ 
28      $\mathcal{Q} := \mathcal{Q} \cup \{ \langle c, this, v \rangle \mid c \in SC_v \}$ ;
29     forall( $y \in Views$ )  $y.wakeValue(\psi(v))$ ;
30 }

```

---

**Fig. 9.** The Template for Non-Injective Domain Views.

variable views. For a view  $y = f(x)$ , when a value  $v$  is removed from the domain of  $x$ , it is no longer sufficient to just use the **map** function. The view must now decide whether the value  $f(v)$  is still supported for  $y$ . Moreover, if we have a view  $z = g(y)$  and variable  $x$  is trying to decide whether to schedule a constraint involving  $z$ , it must query  $z$  to find out whether the value  $g(f(v))$  is still supported, which depends on whether value  $f(v)$  is still supported in variable  $y$ . Hence, to implement non-injective functions in variable views, waking constraints up must be conditional. It is necessary to implement a method **needToSchedule** on views to determine if the original removal will actually remove a value on the views. Method **wakeValue** now becomes

---

```

1 void wakeValue( $\mathcal{V}$   $v$ ) {
2    $\mathcal{Q} := \mathcal{Q} \cup \{ \langle c, x, x.map(v) \rangle \mid \langle c, x \rangle \in SC_v \ \& \ x.needToSchedule(v) \}$ ;

```

---

The implementation of `needToSchedule` must also be recursive (like the `map` function) to handle the case of views on views. For space reasons, we let readers figure out the details on how to do so correctly and *only note the conceptual simplicity of domain views*.<sup>3</sup>

*Literal Views* Reified constraints are a fundamental abstraction in constraint programming. For instance, In a magic series  $s$  of length  $n$ , every  $s_i$  must satisfy  $s_i = \sum_{j=0}^{n-1} (s_j = i)$ , i.e., it states that  $s_i$  should be the number of occurrences of value  $i$  in  $s$  itself. To implement this behavior, one could rely on auxiliary boolean variables  $b_{ij} \Leftrightarrow s_j = i$ . for every  $i$  and  $j$  in  $0..n-1$  leading to a quadratic number of boolean variables and reified equality constraints. The reification  $b \Leftrightarrow x = i$  can be seen as a non-injective view and Figure 10 describes its implementation. The view uses two methods not described before: Method `isBoundTo(i)` on variable  $x$  holds if  $D(x) = \{i\}$ , while method `bind(i)` succeeds if  $i \in D(x)$  and reduces the domain  $D(x)$  to  $\{i\}$ . With these two functions, the implementation is direct with the methods `member`, `remove`, and `wakeValue` carried out by case analysis on the value of the “reified variable”.

*Modulo Views* We now show a view for a constraint  $y = x \bmod k$  with  $k \in \mathbb{Z}$ . The view implementation maintains the supports for each value  $v \in D(y)$ , i.e.,

$$\forall v \in D(y) \ s_v = \{w \mid w \in D(x) \wedge w \bmod k = v\}$$

Figure 11 depicts a sketch of a simple implementation.

## 7 Monotone and Anti-Monotone Views

We briefly mention how to exploit monotone and anti-monotone properties to perform additional operations such as `updateMin` and `updateMax`. These techniques are well-known and are only reviewed here for completeness.

**Definition 3 (Monotone/AntiMonotone Function)** *An injective function  $\psi$  is monotone if  $\forall v, w : v \leq w \rightarrow \psi(v) \leq \psi(w)$ . It is anti-monotone if  $\forall v, w : v \leq w \rightarrow \psi(v) \geq \psi(w)$ .*

If  $\psi : \mathbb{Z} \rightarrow \mathbb{Z}$  is a monotone function and  $y$  is a view on  $x$ , then the update operations on bounds becomes

---

```
1 bool updateMin( $\mathbb{Z}$  v) { return x.updateMin( $\psi^{-1}(v)$ ); }
2 bool updateMax( $\mathbb{Z}$  v) { return x.updateMax( $\psi^{-1}(v)$ ); }
```

---

ignoring the case where  $\psi^{-1}(v)$  is not well-defined. When  $\psi$  is anti-monotone, they become

---

```
1 bool updateMin( $\mathbb{Z}$  v) { return x.updateMax( $\psi^{-1}(v)$ ); }
2 bool updateMax( $\mathbb{Z}$  v) { return x.updateMin( $\psi^{-1}(v)$ ); }
```

---

<sup>3</sup> Method `needToSchedule` must also update any internal state of the views. From a semantic standpoint, it would desirable to have another recursive method to notify the view that value  $v$  has been removed and to update the state.

---

```

1 implementation ReifiedDomainView
2    $\mathcal{X}$   $x$ ;
3    $\{C\}$   $SC$ ;
4    $\{C\}$   $SC_v$ ;
5    $\{\mathcal{X}\}$   $Views$ ;
6    $\mathbb{Z}$   $i$ ;
7 DomainReifiedView( $\mathcal{X}$   $\mathcal{X}$ ,  $\mathbb{Z}$   $i$ ) {
8    $SC := \emptyset$ ;  $SC_v := \emptyset$ ;  $Views := \emptyset$ ;  $i := i$ ;
9 }
10 bool member( $\mathbb{Z}$   $v$ ) {
11   if  $v = 0$  return  $\neg x.isBoundTo(i)$ ;
12   else return  $x.member(i)$ ;
13 }
14 bool remove( $\mathbb{Z}$   $v$ ) {
15   if  $v = 0$ 
16     return  $x.bind(i)$ ;
17   else return  $x.remove(i)$ ;
18 }
19 ...
20 void wakeValue( $\mathcal{V}$   $v$ ) {
21   if  $v = i$ 
22      $\mathcal{Q} := \mathcal{Q} \cup \{ \langle c, this, 1 \rangle \mid c \in SC_v \}$ ;
23     forall ( $y \in Views$ )  $y.wakeValue(1)$ ;
24   else
25     if  $\neg member(0)$ 
26        $\mathcal{Q} := \mathcal{Q} \cup \{ \langle c, this, 0 \rangle \mid c \in SC_v \}$ ;
27       forall ( $y \in Views$ )  $y.wakeValue(0)$ ;
28 }

```

---

**Fig. 10.** A Domain View for Reified Views.

## 8 Empirical Evaluation

We now describe experimental results to demonstrate the efficiency of domain views. The experiments were run on MacOS X 10.8.3 running on a Core i7 at 2.6Ghz, using the OBJECTIVE-CP optimization system [11]. The complete implementation of the integer and boolean variables, along with their domain and their views (including literal views) is around 3,200 lines of code, which is similar to the type of code reuse advertised for Gecode [9]. OBJECTIVE-CP pushes the methodology advocated in [9] to the limit, only supporting core constraints and using views to obtain more complex versions. For instance, the CP solver in OBJECTIVE-CP provides  $\sum_{i=0}^n x_i \leq b$  but not  $\sum_{i=0}^n a_i \cdot x_i \leq b$ . Note that cost-based propagation for COP would, of course, mandate global constraints retaining the  $a_i$ . OBJECTIVE-CP supports value-based propagation and non-injective views, which demonstrates the additional functionalities provided by domain views. Note that the experiments only aim at demonstrating the practicability of domain views: See [9] for the benefits of views.

---

```

1 implementation ModuloDomainView
2   ...
3   int k;
4    $\{\mathbb{Z}\}[]$  S;
5 DomainReifiedView( $\mathcal{X}$   $x$ ,  $\mathbb{Z}$   $k$ ) { ... }
6 bool member( $\mathbb{Z}$   $v$ ) { return  $S_v \neq \emptyset$ ; }
7 bool remove( $\mathbb{Z}$   $v$ ) {
8   forall ( $w \in S_v$ )
9     if  $\neg x.remove(w)$  return false;
10  return true;
11 }
12 ...
13 void wakeValue( $\mathcal{V}$   $w$ ) {
14    $v := w \bmod k$ ;
15   if  $\neg \text{member}(v)$ 
16      $\mathcal{Q} := \mathcal{Q} \cup \{ \langle c, this, v \rangle \mid c \in SC_v \}$ ;
17     forall ( $y \in Views$ )
18        $y.wakeValue(v)$ ;
19 }

```

---

**Fig. 11.** A Domain View for a Modulo Function.

*Benchmarks* The implementation was validated with on a variety of benchmarks relying on views. The experiments compare implementations with no views, with the optimized variable views (with subtype polymorphism), and domain views. When no-views are used, the implementation uses the constraints and auxiliary variables introduced during the flattening of the model. The implementation uses the same models throughout and the search space and pruning are always identical. For **bibd**, we follow [9] and rewrite the boolean relations  $a \wedge b$  as  $\neg(\neg a \vee \neg b)$  to ensure that the system uses negation views. Specifically, **knapsack** use linear equations  $\sum_{i \in S} x_i = b$  and introduce views for the coefficients. The Steel Mill **Slab** problem relies on literal views for the color constraint on slab  $s : \sum_{c \in Colors} \vee_{o \in Orders[c]} (x_o = s) \leq 2$ . **Debruijn** uses both linear equations as well as reifications. **Langford** uses affine views to “shift” indices within element constraints. **Magicseries** clearly relies on reifications. **Sport** is the classic sport scheduling benchmark and uses global constraints.

*Measurements* The benchmarks use a simple first-fail heuristic as decomposition may change the behavior of more advanced heuristics (e.g., WDEG) and these experiments are only interested in assessing view implementations, not inherent speed. Table 1 offers a comparative view of the results. It is based on 50 execution of each benchmark to account for the inherent variability related to modern processor technology. Columns  $\mu(T_{cpu})$  and  $\mu(T_{wc})$  give the average user-time or wall-clock times in milliseconds. Columns  $\sigma(T_{cpu})$  and  $\sigma(T_{wc})$  report the standard deviations for those run times. Column  $|M|$  reports the peak memory consumption in kilobytes for the entire process. The measurement was taken at the level of the **malloc** C-runtime function and includes all memory

<i>Bench</i>	type	$\mu(T_{cpu})$	$\mu(T_{wc})$	$\sigma(T_{cpu})$	$\sigma(T_{wc})$	$ M (KB)$	$P.(\times 1000)$
bibd(6)	NO-VIEW	1,088.4	1,130.5	222.1	227.9	44,652	1,984
bibd(6)	DOMAIN-VIEW	729.8	759.2	107.6	111.0	29,197	804
bibd(6)	VAR-VIEW	644.7	671.2	59.4	60.2	28,082	805
knapsack(4)	NO-VIEW	8,857.5	8,873.9	180.9	184.8	987	33,207
knapsack(4)	DOMAIN-VIEW	6,768.0	6,784.5	166.8	176.1	812	2,949
knapsack(4)	VAR-VIEW	6,151.2	6,164.5	109.2	111.0	789	3,062
ais(30)	NO-VIEW	1,341.8	1,348.9	52.2	57.6	1,336	2,734
ais(30)	DOMAIN-VIEW	1,355.3	1,361.5	31.8	32.4	1,336	2,734
ais(30)	VAR-VIEW	1,354.9	1,362.4	26.1	26.5	1,337	2,734
sport	NO-VIEW	4,851.7	4,864.2	111.3	116.0	2,030	3,361
sport	DOMAIN-VIEW	4,850.9	4,864.3	179.4	189.3	2,029	3,361
sport	VAR-VIEW	4,936.1	4,949.5	231.8	236.7	2,029	3,361
langford(9/3)	NO-VIEW	6,060.5	6,076.1	298.8	306.2	1,375	54,027
langford(9/3)	DOMAIN-VIEW	7,008.3	7,026.6	316.8	323.5	1,368	56,893
langford(9/3)	VAR-VIEW	6,859.5	6,877.9	242.7	248.5	1,366	56,887
debruijn(2/12)	NO-VIEW	7,665.5	8,437.3	153.4	169.2	624,635	2,558
debruijn(2/12)	DOMAIN-VIEW	7,292.2	8,014.2	111.2	144.1	552,515	946
debruijn(2/12)	VAR-VIEW	6,935.2	7,628.2	384.9	422.4	550,792	967
slab	NO-VIEW	4,845.2	4,919.4	108.6	115.4	84,092	4,403
slab	DOMAIN-VIEW	2,243.5	2,294.0	128.7	138.8	51,725	909
slab	VAR-VIEW	4,509.8	4,578.7	94.7	99.4	73,929	2,968
magicserie(300)	NO-VIEW	17,951.2	18,164.7	284.0	300.8	231,622	30,771
magicserie(300)	DOMAIN-VIEW	8,318.3	8,443.6	191.9	201.4	122,026	257
magicserie(300)	VAR-VIEW	14,879.9	15,088.0	429.1	441.1	229,288	20,745

**Table 1.** Experimental Results on Variable and Domain Views.

allocations done by the executable. Finally, column  $P$ . reports the number of propagation events recorded by the engine (in thousands).

Without surprise, the results indicate that a minimalist kernel *must* use views to be competitive. The differences in memory consumptions and running times are often quite significant when contrasted with view-based implementations. For all benchmarks involving only injective views, variable and domain views are essentially similar in time and space efficiency. Given the standard deviations, the differences in efficiency are not statistically significant, although variable views are often slightly more efficient. This is not always the case, as the sport-scheduling problem indicates. The main benefit of domain views is to support non-injective views simply and efficiently. This is particularly clear on the benchmarks relying on reifications, i.e, **slab** and **magicserie**. The benefits are in terms of runtime and memory consumption. The runtime benefits are quite substantial, as the running time is halved on the Steel Mill Slab problem. The dramatic drop in the number of propagations is easily explained by the absence of constraints of the form  $b \Leftrightarrow (x = v)$ , yet, the same work is still carried out by the view, albeit at a much lower overhead.

In summary, the experimental results show that domain views do not add any measurable overhead on injective views and bring significant benefits on non-injective views, which they support elegantly.

## 9 Related Work

It is important to contrast the variable and domain view implementations proposed here with another approach using delta-sets and advisors [7,9]. Advisors are another way of “simulate” value-based propagation.<sup>4</sup> An advisor is associated with a variable and a constraint and it modifies the state of the constraint directly upon a domain modification for its variables. Advisors do not go through the propagation queue but modify the state of their constraint directly. This has both an advantage (speed) and an inconvenience, since an advisor may be called while its constraint is propagating; Hence some care must be exercised to maintain a consistent state. Advisors also receive the domain change (called a delta set) which they may query.

Advisors can be associated with variable views. The view must now be upgraded to query, not only the domain, but also the delta sets. In other words, the queries on the delta must transform the domain delta, say  $\{v_1, \dots, v_n\}$ , through the view to obtain  $\{\phi(v_1), \dots, \phi(v_n)\}$ . Gecode [9] does not compute delta sets exactly but approximates them by intervals instead. A complete implementation of value-based propagation would require the creation of these delta sets. Advisors and delta sets can be used in the case of non-injective functions but that solution would still go through the propagation queue and use a constraint. Indeed, by design, advisors do not propagate constraints.

The key advantage of domain views in this context is their ability to implement non-injective views without going through the propagation queue.

## 10 Conclusion

This paper reconsidered the concept of views, an important abstraction provided by constraint-programming systems to avoid the proliferation of constraints, while preserving the efficiency of dedicated implementation. It proposed an alternative to the concept of variable views, typically featured in constraint-programming systems. Contrary to variable views, domain views only delegate domain operations and maintains their own set of constraints to watch. Domain views simplify the implementation of constraint-programming systems featuring value-based propagation as they avoid manipulating first-order functions (or objects implementing a similar functionality). They also make it possible to implement, in simple ways, views featuring non-injective functions. These are particularly useful for reified constraints, which are also an important features of constraint-programming systems. Experimental results demonstrate that domain views introduce a negligible overhead (if any) over variable views and that views over non-injective functions, which are elegantly supported by domain views, provide significant benefits.

---

<sup>4</sup> It is only a simulation since an advisor updates the constraint state but does not propagate a constraint itself. They are second-class citizens by choice in Gecode [7].

## References

1. H. P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.
2. N. Beldiceanu, M. Carlsson, S. Demassey, and T. Petit. Global constraint catalogue: Past, present and future. *Constraints*, 12(1):21–62, Mar. 2007.
3. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. H. Hartel, and H. Kuchen, editors, *PLILP*, volume 1292 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 1997.
4. I. Dynadec. Comet v2.1 user manual. Technical report, Providence, RI, 2009.
5. P. V. Hentenryck, V. Saraswat, and Y. Deville. Constraint processing in cc(fd). Technical report, 1992.
6. Ilog Solver 4.4. Reference Manual. Ilog SA, Gentilly, France, 1998.
7. M. Lagerkvist and C. Schulte. Advisors for incremental propagation. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, Sep 2007.
8. C. Schulte and G. Tack. Perfect derived propagators. In P. J. Stuckey, editor, *CP*, volume 5202 of *Lecture Notes in Computer Science*, pages 571–575. Springer, 2008.
9. C. Schulte and G. Tack. View-based propagator derivation. *Constraints*, 18(1):75–107, 2013.
10. P. Van Hentenryck, Y. Deville, and C. Teng. A Generic Arc Consistency Algorithm and Its Specializations. *Artificial Intelligence*, 57(2-3), 1992.
11. P. Van Hentenryck and L. Michel. The OBJECTIVE-CP Optimization System. In *Proceedings of the 19<sup>th</sup> International Conference on Principles and Practice of Constraint Programming*, Sep 2013.